LA-UR-19-26842
Approved for public release; distribution is unlimited.

Title: Checkpoint-Restart and Platform Design Criteria

Author(s): Loncaric, Josip

Intended for: Technical report

Issued: 2019-07-17

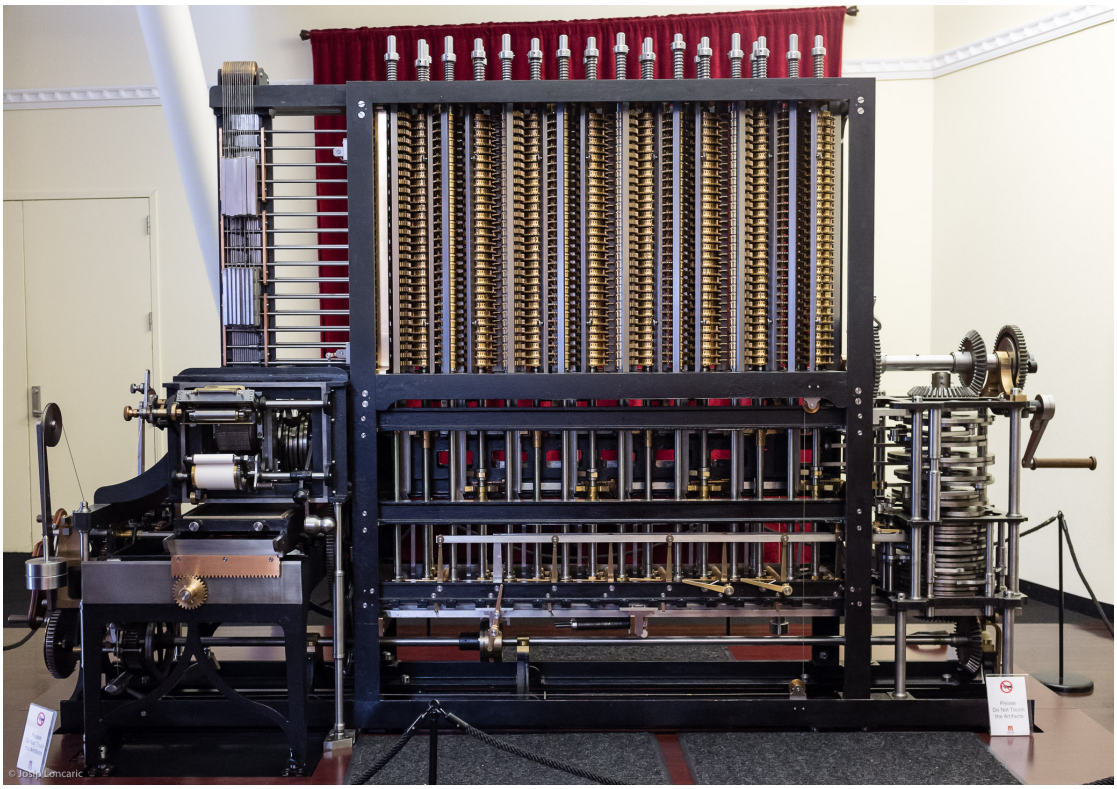
Checkpoint-Restart and Platform Design Criteria

Josip Loncaric, LANL
July 12th, 2019

Abstract: Long computations often use checkpoint/restart to survive the inevitable failures. Optimal checkpoint/restart strategy can be derived in terms of special functions, and approximated within the regime of interest. As a further step under nominal assumptions, platforms can be optimized for maximum progress rate, computed in terms of non-dimensional parameter JMTTI/Delta, where JMTTI is the Job Mean Time To Interrupt and Delta is the time it takes to write (or read) a checkpoint. Scaling to a larger platform of the same technology requires scaling checkpoint bandwidth quadratically to maintain the same progress rate. This was observed during the design of LANL's Roadrunner in 2007, led to burst buffers proposed in 2009 and introduced on Trinity in 2015. The need to compute reliably continues to impact the design of future exascale platforms.

Introduction

Correctness has been the primary driver of computer development, more important than performance. Nobody cares how fast incorrect results can be obtained. Any computation is a physical, error-prone process, whether performed by humans or by machines. Even digital electronics is only an approximation of mathematical logic, dependent on carefully constructed statistical and analog physical processes.



A working replica of Charles Babbage' s Analytical Engine, at the Computer History Museum, Mountain View, CA. Photographed by the author in 2013.

In the early 19th century, published mathematical and navigation tables (computed and typeset by humans) contained a lot of errors, endangering naval operations, so Charles Babbage conceived and designed his "analytical engine" mechanical computer to achieve the "unerring certainty" of machinery. The Babbage analytical engine was not fully built until the 21st century, when his original objectives were finally realized. Nevertheless, while machine computation improves reliability, it is not perfect, particularly if one relies on noisy analog processes. That's why in 1948 Project Whirlwind proposed digital computation using vacuum tubes, and tested correctness of this approach at MIT by building a 5-bit multiplier. Experience demonstrated that even a digital computer could make errors, usually at night when a janitor used an elevator, causing power glitches. MIT solved this problem by deploying a rotary uninterruptible power supply (RUPS), a solution which sometimes gets used to this day to improve power quality for critical equipment.

Norman H. Taylor examines the Whirlwind 5-digit multiplier (October 21, 1948). This multiplier was used to verify correctness of vacuum tube digital computation. Image from The MITRE Corporation photo archive, originally at http://www.mitre.org/about/photo_archives/whirlwind_photo.html but no longer available.

Over the centuries since Babbage's mechanical computer, technology has dramatically improved. Today's CMOS circuitry is extremely reliable, but demands for computation have become extreme as well. Due to exponential growth in the demand for computation, one must deal with tyranny of exponents: while today's technology can perform over 10^{20} operations between failures, physical effects can and do cause faults in electronics, which can propagate to become errors in the computation. There are many reasons for this, including cosmic rays, natural radiation, noise, voltage glitches, thermal effects, physical deterioration, etc. One must be concerned about computational errors, because we are approaching the point where trillion-dollar decisions rely on billion-dollar computations on extreme scale computers.

LANL's Roadrunner was the first in the world to exceed 1 PF/s sustained Linpack performance in 2008. Checkpointing to disk based parallel file system was the main resilience strategy on Roadrunner.

In 2008, LANL built Roadrunner, the first petaFLOP computer in the world, capable of executing over 10^{15} double precision floating point operations per second. With the mean time between failures (MTBF) of about a day, Roadrunner could deliver about 10^{20} operations between failures. The same year, DARPA estimated that an improved exascale computer built in 2018 might operate for only 34 minutes between errors, implying that about 2×10^{21} operations could be expected between errors.

Fortunately for applications, modern computers contain a lot of error detection circuitry, so that most errors can be caught and acted upon. Such detection covers only about 90-95% of modern processor die area, mostly in regular structures such as register files and memory hierarchy. Memory and data paths are protected by error correcting codes (ECC) and/or at least parity. All of this can be done at a modest cost, but about 5-10% of complex logic circuitry is too costly to protect by error detection. At small scales in less sensitive applications, that's an acceptable risk, but for high value large scale computations, risks must be anticipated and managed. In 2008, LANL subjected Roadrunner hardware to accelerated testing in the LANSCE neutron beam, and concluded that 5-10% of faults can produce silent data corruption (SDC), where the application unknowingly continues to operate on corrupted data. Given 2008 technology, that meant over 1000 hours of full-scale operation between infrequent SDC events, a rate low enough to be recognized by spotting computational outliers during expert analysis and comparison of results, if by some chance an error produced a significant difference.

Detected error rates are much higher, occurring daily or even hourly. Correctable memory errors may even occur every minute or so on a large platform, but error correcting codes (ECC) are very effective at repairing

single bit errors without interrupting the computation in progress. Multi-bit errors per ECC-protected word are not reliably correctable and can cause application interrupts. Other errors occur in digital logic circuitry, often detected through the machine check exception (MCE) mechanism. Additional interrupts can arise in the high speed network, in storage, in system software, or in the application itself. All of these paths can lead to an application interrupt, and a signal to clean up the situation.

Checkpoint/restart is the most common method applications use to cope with interrupts. The application periodically writes out its state to storage in a separate failure domain, and if an interrupt occurs, restarts from the latest valid checkpoint. This process involves making a number of decisions, such as how often to checkpoint. This has been analyzed by Daly [1], who improved upon the previous analytical solution by developing simple approximations to the optimal checkpoint interval, as a function of a number of parameters. His analysis proceeds by optimizing the following expression for total time to solution for a long running application (Eq. 20 in his paper, adapted to Mathematica):

In[1]:=
```
Tw[Tau_] := Ts Exp[R / JMTTI] (Exp[(Tau + Delta) / JMTTI] - 1) (JMTTI / Tau)
```

Above, Tw is the expected wall clock time to solution including all checkpoint, restart, and rework overheads, as a function of Tau, the chosen compute interval between checkpoints. JMTTI is the job mean time to interrupt, R is the time to restart the job, Delta is the time to write a checkpoint, and Ts is the compute time to solution. This expression is valid for long running jobs, where Delta << Ts applies. Daly computes the exact solution to minimizing Tw as a function of Tau, given in terms of Lambert's function W (also called the product logarithm), then approximating this in the regime of interest by simpler expressions. The following analysis follows an equivalent but slightly different path.

Optimal Checkpoint/Restart Strategy and Progress Rate

Since read and write bandwidths are similar and the saved application state is large, time to restart from a checkpoint R is approximately equal to Delta. The reliable progress rate is Ts/Tw, and that's what we'd like to maximize. Moreover, since JMTTI defines a time scale for failures, it is natural to non-dimensionalize in terms of multiples of JMTTI, introducing non-dimensional parameters x=Delta/JMTTI and y=Tau /JMTTI. The formula for reliable progress rate of a long running application as a function of x and y becomes:

In[2]:=
```
RPR[x_, y_] := y / (Exp[x] * (Exp[x + y] - 1))
```

This function is universal, in the sense that all such system designs produce the same equation, dependent only on x and y. The reliable progress rate is maximized when y = Tau/JMTTI is chosen as follows:

In[3]:=
```
yOpt[x_] := ProductLog[-e^(-1-x)] + 1
```

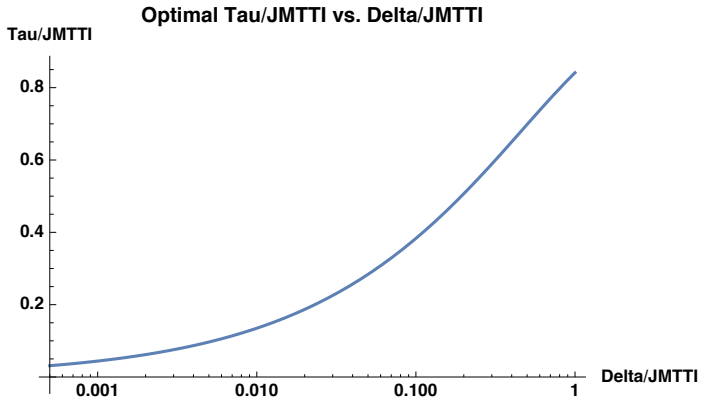
In other words, the optimal compute interval between checkpoints is:

In[4]:=
```
TauOpt[Delta_, JMTTI_] := JMTTI * (1 + ProductLog[-e^(-1-Delta/JMTTI)])
```

This is identical to Daly's exact solution for the R = Delta case, expressed in terms of special function ProductLog[] available in Mathematica. We can plot this non-dimensionalized optimal y = Tau/JMTTI as a function of non-dimensional x = Delta/JMTTI:

In[5]:=
```
LogLinearPlot[yOpt[x], {x, 0.0005, 1},
 PlotLabel → "Optimal Tau/JMTTI vs. Delta/JMTTI",
 AxesLabel → {"Delta/JMTTI", "Tau/JMTTI"}]
```

Out[5]=

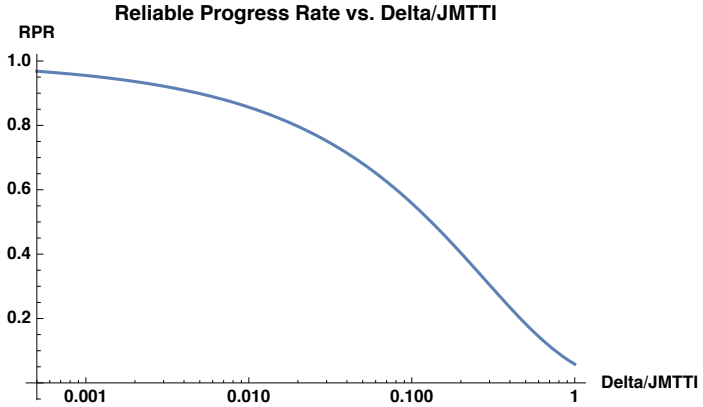


For Delta/JMTTI << 1, checkpointing is inexpensive and can be done several times between interrupts, which do not require much rework on restart. When Delta/JMTTI = 1, checkpointing is expensive and might even fail before it is completed, so it is done about once between interrupts, implying a lot of rework on restart. At that optimal compute interval between checkpoints, the reliable progress rate behaves as follows:

In[6]:=
```
RPRopt[x_] := RPR[x, yOpt[x]]
```

In[7]:=
```
LogLinearPlot[RPRopt[x], {x, 0.0005, 1},
 PlotLabel → "Reliable Progress Rate vs. Delta/JMTTI",
 AxesLabel → {"Delta/JMTTI", "RPR"}]
```

Out[7]=



Reliable progress rate is quite reasonable 90% or better for Delta/JMTTI < 0.005, but as checkpoint times become commensurate with JMTTI, progress rate declines exponentially. For that reason, scaling the platform does not scale reliable performance. In 2007, Loncaric [2] showed that reliable performance reaches a peak and then actually declines if linear scaling is applied. Consider a platform characterized by a particular value of x, and build a k times larger platform, with k times more available performance. The checkpoint time Delta will remain the same, because k times more application state takes about the same amount of time to checkpoint using k times more storage bandwidth. However, the failure rate will increase, because k times more components fail k times more often, so x' = x*k will characterize the scaled platform. The reliable performance gain won't be k, because of increased checkpoint/restart overheads. The scaled platform reliable performance

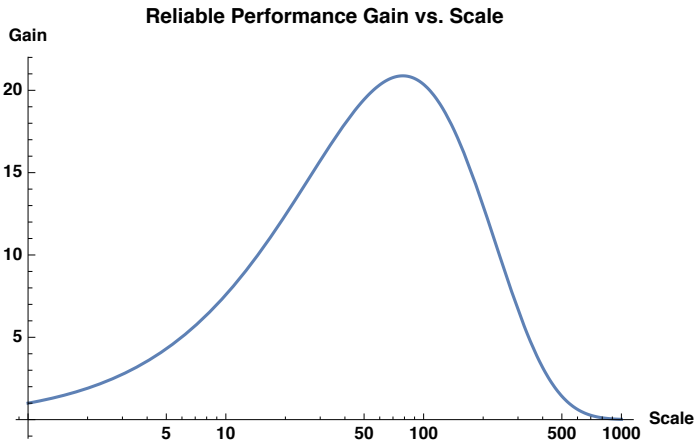
gain over the baseline platform will be:

In[8]:=
```
RPgain[k_, x1_] := k * RPRopt[x1 * k] / RPRopt[x1]
```

Assume that the k=1 platform was designed for about 90% progress rate, i.e. that x1=0.005. Scaling the same technology linearly produces the following outcome:

In[9]:=
```
LogLinearPlot[RPgain[k, 0.005], {k, 1, 1000},
 PlotLabel → "Reliable Performance Gain vs. Scale", AxesLabel → {"Scale", "Gain"}]
```

Out[9]=



Reliable performance rises with platform scale up to a maximum near k=100, then precipitously declines. The maximum can be computed:

In[10]:=
```
FindMaximum[RPgain[k, 0.005], {k, 100.}]
```

Out[10]=
```
{20.8796, {k → 78.4937}}
```

Scaling this baseline platform cannot produce reliable performance more than 20.8796 times greater, because for k > 78.4937 its reliable performance actually starts to decline. While different multiples would be obtained for different baseline platform x1 values, the generic conclusion is that performance scaling cannot overcome exponential loss in reliable progress rate inherent in scaling. Moreover, it is very expensive to buy 78.4937 times more hardware for a gain of only 20.8796 times more reliable performance. This observation was first made in 2007 by Loncaric during Roadrunner development. It was presented at the LANL booth during the SC07 conference [2].

For these reasons, linear platform scaling does not make sense for very large platforms. For example, if we scaled Roadrunner petascale technology (with x1 approximately 0.003) to exascale, we'd pay 1000 times more for no more delivered performance. As one scales a computing platform, checkpointing capabilities cannot scale linearly. One can easily see that to maintain the same progress rate, checkpoint bandwidth must scale quadratically, because k times bigger machine also fails k times more often, so to maintain the same Delta/JMTTI ratio, one needs k^2 times more checkpoint bandwidth (a simple scaling argument by Andy White [3]). This is also unsustainable: eventually, the storage subsystem becomes more expensive than the compute platform itself.

This problem was analyzed by Grider [4] in November 2009, who proposed burst buffers to address checkpoint/restart needs of exascale platforms. This meant switching from storage technology priced for capacity

(hard disks) to technology bought for bandwidth (flash memory). After years of preparation, design, and development, LANL's Trinity platform (deployed in two phases in 2015 and 2016) became the first supercomputer in the world featuring burst buffers.

Explaining the details of checkpoint/restart in a request for proposals (RFP) isn't appropriate, because brevity and clarity are required. Therefore, in 2012 Loncaric and Doerfler developed the JMTTI/Delta criterion [5] which has been used to procure Trinity [6,7] and subsequent platforms. The rationale is that since reliable progress rate depends nonlinearly on x = Delta/JMTTI, as long as the proposed platform delivers x in a good range, the platform will be productive enough. For RFP requirements, it's easier to work with 1/x, which is equivalent (e.g. requiring JMTTI/Delta > 200 instead of Delta/JMTTI < 0.005). Trinity burst buffer requirements specified JMTTI/Delta > 200, which suffices to get progress rate of about 90% or better (subsequent balancing of costs and risks adjusted this somewhat). This criterion is very easy to interpret and verify. As long as the vendor can build a platform that's reliable enough, or deliver high enough checkpoint bandwidth to meet this criterion, the platform should be able to achieve the desired reliable performance even on the largest problems.

Trinity, the first supercomputer in the world with burst buffers, 2015. Burst buffers enable efficient and reliable computation at extreme scales.

Approximations Useful in Practice

Exact analytic solutions can be evaluated in Mathematica, but no commonly available tool has the required special functions. While interpolations are possible in Excel based on Mathematica computed lookup tables, in practice it is beneficial to work with simplified approximations applicable near the regime of interest, where Delta << JMTTI and x approaches zero. Simple expressions can be derived, and as long as it is understood that they apply for small enough x. For x < 0.03, the reliable progress rate is 75% or better. In this regime, the reliable progress rate at optimal y can be approximated within 1% as follows:

In[11]:=
```
RPRapprox[x_] := 1 - Sqrt[2 * x] - x / 3
```

Daly's approximation for TauOpt is equivalent to the following approximation of optimal y:

In[12]:=
```
yApprox[x_] := Sqrt[2 * x] - x
```

Both of these expressions are approximations which become rather inaccurate for x > 0.03 or so. For small enough x, approximations can be very useful to analyze cost, risk, and performance tradeoffs in designing a platform.

Multi-level Checkpointing

By introducing burst buffers, Trinity created an opportunity for multi-level checkpointing. Due to its large memory (over 2 PiB), even checkpointing to the burst buffer could take up to 10 minutes. Trinity also has a disk based parallel file system, which could take almost 30 minutes to checkpoint Trinity memory (smaller application states can be checkpointed faster). The difference isn't large due to risk mitigation: Trinity had to have a capable PFS, just in case burst buffers could not be made to perform as expected. This conservative approach anticipated x = 0.007 for checkpointing to the burst buffer, and x = 0.02 for checkpointing to the disk based parallel file system, resulting in anticipated progress rates of 88% and 79% respectively. Trinity can perform successfully even without burst buffers. By now, with burst buffer capabilities demonstrated, a less conservative approach can be taken on future platforms.

It should be noted that the burst buffer space on Trinity is allocated to jobs logically, not physically. Trinity burst buffer is globally accessible, because it is contained in burst buffer nodes on the same high speed network as the compute nodes. This flexibility is useful, but may have to be changed in the future, as growth in bandwidth requirements forces more and more burst buffer nodes to share this bandwidth. Ultimately, every node may need to host a portion of the burst buffer, implying more complex management of space and bandwidth allocations between jobs. A step in that direction was made with subsequent platforms (Sierra and Summit) where this work continues. Regardless of how burst buffers are implemented, they too can fail, so secondary checkpoints to disk based PFS remain necessary.

In multi-level checkpointing, the first level (to a burst buffer) protects against a loss of application state, and the second level (to disks) protects against a loss of burst buffer data. A simple way to think about this can be illustrated as follows. A large application may perform optimally when one checkpoints to the burst buffer every 3 hours. Assume that the mean time to burst buffer data loss is 1000 hours, and that it takes an hour to write out application state from disk to the PFS. Using the above equations, the optimal way to protect against loss of state in the burst buffer is to write it out to the PFS every 44 hours, e.g. approximately every 15^{th} burst buffer checkpoint needs to be saved to the PFS. This is quite feasible. In fact, because applications use the saved state for later analysis, burst buffer checkpoints are written to disk more frequently, albeit at a more leisurely rate because data transfers between the burst buffer and PFS need not delay application progress. Checkpointing to the burst buffer is on the critical path, but saving checkpoints to disk isn't, as long as it can overlap with the compute interval between checkpoints.

Cost optimization

During Trinity design, costs were not known, but costs became known during final negotiations, so it was possible to compute the cost constrained design with optimal reliable progress rate, if everything worked as expected. However, there are no guarantees that advanced technology not demonstrated before will work as expected. Therefore, this calculation was modified by perceived risks, in order to obtain a productive platform whether or not Trinity burst buffers delivered the expected benefits. Future platforms can take advantage of cost optimization with more confidence, because some risks have been reduced by demonstration on Trinity.

A simple way to approach this problem is in the iso-cost context, optimizing what fraction of the cost is spent on compute, on burst buffer bandwidth, and on PFS capacity. Because cost fractions add up to 100%, there are only two design parameters, and a cost model can be optimized numerically, subject to operational constraints such as minimum capacity and performance levels. Numerical calculations performed to date suggest that more productive platforms can be obtained by spending more on burst buffer bandwidth and bit less on PFS capacity. Specifics will depend on future cost data, usually not disclosed for proprietary reasons, but in the final negotiations this kind of design optimization can prove beneficial and should lead to more productive platforms.

Conclusion

Optimal checkpoint/restart policies lead to the best possible reliable progress rate for a calculation. A productive platform design requires the Delta/JMTTI ratio to be small enough, typically less than 1/200 for at over 90% reliable progress rate. The inverse of this ratio JMTTI/Delta has been successfully used to procure Trinity and subsequent platforms, without overly complicating the RFP requirements.

Multi-level checkpoint restart strategies are possible when there are multiple tiers of storage, such as flash and disk. The key insight is that the outer level of checkpointing (to disk) protects against failures in the inner layer of checkpointing (to flash), which in turn protects against application interrupts. While Trinity burst buffer is fairly large, it is smaller and more reliable than the compute platform itself. Because the burst buffer mean time to data loss is about a month, infrequent checkpointing to disk suffices to protect against that risk.

Trinity burst buffer was built using a globally accessible pool of flash storage. However, as platforms are scaled even further, and bandwidth requirements continue to grow quadratically, the number of burst buffer nodes on the network will have to grow. Half of the network ports may be needed to accept checkpointing bandwidth while the other half supplies the data. Ultimately, if storage is provided on every node, nodes could be writing checkpoints to other nodes, leveraging the bi-directional capability of the high speed network. This situation will create challenges in managing storage and jobs, particularly permissions of who has access to what when, but designs of this type have already appeared on Summit (ORNL) and Sierra (LLNL) platforms.

Checkpoint/restart will eventually become insufficient to deliver good progress at extreme scales, because quadratic growth in bandwidth requirements is only temporarily helped by faster storage technology. Other

difficulties should be expected, such as the ability to detect errors in a timely manner before locality of error influence is lost. If error detection is delayed or absent, the checkpoint/restart model will have to be changed and optimized differently. Fine-grained protection (e.g. at a transaction level) may be needed to be implemented in hardware, and error detection coverage improved. There is a growing tension between performance, power efficiency, and reliability. As technologies evolve, platform optimization must remain focused on the delivered reliable performance, while respecting design constraints such as budgets.

References

[1] "A higher order estimate of the optimum checkpoint interval for restart dumps," Daly, John T. ; LA-UR-04-0501 ; 2004. Published in Future Generation of Computer Systems 22 (2006) 303-312. https://www.sciencedirect.com/science/article/pii/S0167739X04002213?via%3Dihub

[2] "Beyond Petascale: The Looming Crisis in Checkpoint/Restart Approach to Fault Tolerance," Loncaric, Josip, LA-UR-07-07278 ; 2007. Presented at the SC07 LANL booth. http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-07-07278

[3] Andy B. White, personal communication, Nov. 8, 2007.

[4] "Exa-Scale FSIO Analysis," Gary Grider, personal communication, Nov. 2009.

[5] "Trinity RAS and Productivity Metrics: Platform Efficiency and Burst Buffer," Loncaric, Josip; Doerfler, Doug; internal presentation to the Trinity/NERSC-8 team, Oct. 4, 2012.

[6] "Trinity and NERSC-8 Computing Platforms / Request for Proposal (RFP) No. 241314," Knox, Darren ; Aug. 7, 2013. Posted to LANL business opportunities site at the time, now gone.

[7] "Trinity Technical Requirements Document," Lujan, James W. ; LA-UR-13-26215 ; 2013-08-06. https://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-13-26215